Informe de paralelización métodos de itinerarios

Samuel Escobar Rivera - 2266363
Joseph David Herrera Libreros - 2266309
Juan David Cuellar López - 2266087

Carlos Andres Delgado Savaadera

Universidad del Valle Sede Tuluá
Facultad de ingeniería
Ingeniería en Sistemas

Funcionamiento del método ItinerariosPar:

El método ItinerariosPar acoge una lista de vuelos y una lista de Aeropuertos que retorna una función que recibe dos cadenas de String, los cuales son los códigos de los aeropuertos de origen y de destino. Para la paralelización de los métodos usamos future, dado que es un tipo de objeto que representa un valor disponible ahora o que posiblemente se tenga en un futuro, esto nos permite manejar las tareas de forma asincrónica y concurrente sin bloquear el hilo principal.

```
if (origen == destino) {
  Future.successful(List(rutaActual))
```

La función auxiliar buscar itinerario primero verifica si el aeropuerto de origen es igual al aeropuerto de destino y si se cumple la condición retorna la lista con el itinerario de vuelos de tipo future, lo que significa que ya existe un valor o itinerario completo correcto y no se necesita ejecutar otro calcular de manera asincrónica.

```
    } else if (visitados.contains(origen)) {
      Future.successful(List())
    } else {
      val vuelosDesdeOrigen = vuelosPorOrigen.getOrElse(origen.toLowerCase, List())
      val futurosItinerarios = vuelosDesdeOrigen.map { vuelo =>
        buscarItinerarios(vuelo.Dst.toLowerCase, destino.toLowerCase, visitados + origen, rutaActual :+ vuelo)
      }

      Future.sequence(futurosItinerarios).map(_.flatten)
    }
  }

  (cod1: String, cod2: String) => {
    val futurosResultados = buscarItinerarios(cod1.toLowerCase, cod2.toLowerCase, Set(), List())

    import scala.concurrent.Await
    import scala.concurrent.duration._
```

Aquí verificamos que el itinerario tome aeropuertos visitados, si dado el caso el itinerario toma un aeropuerto ya visitado retornamos una lista vacía para evitar ciclos.

Para cada vuelo desde el aeropuerto de origen, se llama recursivamente a buscaritinerarios para el destino de ese vuelo. Future.sequence convierte la lista de future en un future de lista, esto ejecuta todas las búsquedas recursivas en diferentes hilos y map(_.flatten) combina los resultados de todos los futuros.

Finalmente, future.sequence combinará los resultados y Await.result esperará a que todos los futuros se completen, devolviendo la lista de itinerarios válidos.

Funcionamiento del método ItinerariosTiempoPar:

Para la paralelización del método ItinerariosTiempoPar lo que realizamos fué:

```
(cod1: String, cod2: String) => {
  val itinerariosPosibles = buscarItinerariosFn(cod1, cod2)

  val futurosDuraciones = itinerariosPosibles.map { itinerario =>
    Future {
      calcularDuracionTotal(itinerario) -> itinerario
    }
  }
  val futureResultados = Future.sequence(futurosDuraciones)
  val resultados = Await.result(futureResultados, 10.seconds)
  val itinerariosConDuracion = resultados.map { case (duracion, itinerario) =>
    itinerario
  }
```

Por lo tanto, se crea un futuro para cada itinerario en itinerariosPosibles, que calcula la duración total del itinerario y la asocia con el itinerario.

Necesitamos convertir una lista de futuros en uno solo que incluya una lista de resultados. Esto se hace utilizando ***Future.sequence***. Se toma una lista de futuros, ***Future.sequence*** devuelve un futuro que se completa cuando todos los futuros de la lista se completan, incluyendo una lista de resultados.

Por último, usamos ***Await.result*** para obtener los resultados de los futuros, que bloquea hasta que el futuro completo se haya completado o hasta que se alcance un tiempo de espera predeterminado. ***Await.result*** toma un futuro y un tiempo de espera, y bloquea la ejecución del hilo actual hasta que el futuro o el tiempo de espera se completen. Se espera un máximo de 10 segundos en este caso.

Funcionamiento del método ItinerariosEscalaPar:

Para la paralelización del método **ItinerariosEscalaPar** es prácticamente lo mismo que la función anterior de **ItinerariosTiempoPar**:

```
(cod1: String, cod2: String) => {
  val itinerariosPosibles = buscarItinerariosFn(cod1.toLowerCase, cod2.toLowerCase)

  val futurosEscalas = itinerariosPosibles.map { itinerario =>
    Future {
      calcularNumeroEscalas(itinerario) -> itinerario
    }
  }

  val futureResultados = Future.sequence(futurosEscalas)
  val resultados = Await.result(futureResultados, 10.seconds)
  val itinerariosConEscalas = resultados.map { case (escalas, itinerario) =>
    itinerario
  }
```

Future permite la paralelización al calcular las escalas de cada itinerario de manera asíncrona y en paralelo. Se crea un futuro para cada itinerario en itinerarios posibles que determina el número de escalas. Esto permite la ejecución simultánea de cada cálculo, lo que mejora el rendimiento y la eficiencia.

Tomando una lista de futuros, ***Future.sequence*** devuelve un futuro que se completa cuando todos los futuros de la lista se completan, incluyendo una lista de resultados. La ejecución de tareas paralelas se sincroniza así.

Await.result se bloquea hasta que se completen los futuros resultados o se agote el tiempo de espera de 10 segundos. Esto permite obtener resultados sincronizados de todos los cálculos asíncronos.

Funcionamiento del método ItinerariosAirePar:

Para la paralelización del método **ItinerariosAirePar** es prácticamente lo mismo que las funciones anteriores:

```
(cod1: String, cod2: String) => {
  val itinerariosPosibles = buscarItinerariosAire(cod1, cod2)

  val futurosDuraciones = itinerariosPosibles.map { itinerario =>
    Future {
      calcularDuracionTotal(itinerario) -> itinerario
    }
  }



  val futureResultados = Future.sequence(futurosDuraciones)
  val resultados = Await.result(futureResultados, 10.seconds)
  val itinerariosConDuracion = resultados.map { case (duracion, itinerario) =>
    itinerario
  }

  itinerariosConDuracion.sortBy(calcularDuracionTotal).take(3)
}
```

Future permite la paralelización, lo que permite calcular la duración total de cada viaje de manera asíncrona y en paralelo. Se crea un futuro para cada itinerario en itinerarios posibles, lo que determina la duración total del itinerario. Esto permite la ejecución simultánea de cada cálculo, lo que mejora el rendimiento y la eficiencia.

Tomando una lista de futuros, ***Future.sequence*** devuelve un futuro que se completa cuando todos los futuros de la lista se completan, incluyendo una lista de resultados. La ejecución de tareas paralelas se sincroniza así.

Await.result se bloquea hasta que se completen los futuros resultados o se agote el tiempo de espera de 10 segundos. Esto permite obtener resultados sincronizados de todos los cálculos asíncronos.

Funcionamiento del método ItinerariosSalidaPar:

```
  (origen: String, destino: String, horaCita: Int, minCita: Int) => {
    val tiempoCita = convertirAMinutos(horaCita, minCita)
    val todosItinerarios = buscarItinerariosFn(origen, destino)

    // Filtrar itinerarios de forma concurrente
    val futurosItinerariosValidos = Future.traverse(todosItinerarios) { itinerario =>
      Future {
        if (esValido(itinerario, tiempoCita)) Some(itinerario) else None
      }
    }

    // Convertir la lista de futuros en un futuro de lista
    val futurosResultados = futurosItinerariosValidos.map(_.flatten)

    // Aquí es donde hacemos el bloqueo para obtener el resultado
    val itinerariosValidos = Await.result(futurosResultados, 10.seconds)

    // Ordenar itinerarios de forma concurrente
    val futurosItinerariosOrdenados = Future {
      itinerariosValidos.sortBy { it =>
        val horaLlegada = calcularHoraLlegadaTotal(it)
        val lapsoTiempo = calcularLapsoTiempo(horaLlegada, tiempoCita)
        (lapsoTiempo, calcularHoraSalidaTotal(it))
      }
    }

    // Obtener el resultado ordenado
    val itinerariosOrdenados = Await.result(futurosItinerariosOrdenados, 10.seconds)

    itinerariosOrdenados.headOption.getOrElse(List.empty)
  }
}
```

Para paralelizar el método de itinerario salida primero convertimos el tiempo de la cita en minutos, después aplicamos future.traverse a una función a cada elemento de la lista y devuelve una lista future y cada itinerario se valida de manera paralela para verificar si llega antes de la hora de la cita.

```
def esValido(itinerario: List[Vuelo], tiempoCita: Int): Boolean = {
  val horaLlegada = calcularHoraLlegadaTotal(itinerario)
  horaLlegada <= tiempoCita || (horaLlegada < 1440 && tiempoCita < horaLlegada)
}
```

```
val futurosResultados = futurosItinerariosValidos.map(_.flatten)

// Aquí es donde hacemos el bloqueo para obtener el resultado
val itinerariosValidos = Await.result(futurosResultados, 10.seconds)
```

Es una lista future y map(_.flatten) convierte la lista de opciones en una lista simple de itinerarios válidos.

Await.result bloquea el hilo principal hasta que futurosResultados esté completo o hasta que pasen 10 segundos y obtiene los itinerarios válidos.

```
val futurosItinerariosOrdenados = Future {
  itinerariosValidos.sortBy { it =>
    val horaLlegada = calcularHoraLlegadaTotal(it)
    val lapsoTiempo = calcularLapsoTiempo(horaLlegada, tiempoCita)
    (lapsoTiempo, calcularHoraSalidaTotal(it))
  }
}

// Obtener el resultado ordenado
val itinerariosOrdenados = Await.result(futurosItinerariosOrdenados, 10.seconds)

itinerariosOrdenados.headOption.getOrElse(List.empty)
```

Se ordenan los itinerarios válidos en otro future para no bloquear el hilo principal y se ordenan por la hora de llegada más próxima a la hora de la cita y luego a la hora de salida más tarde.

```
val itinerariosOrdenados = Await.result(futurosItinerariosOrdenados, 10.seconds)

itinerariosOrdenados.headOption.getOrElse(List.empty)
```

Await.result se usa nuevamente para obtener los resultados ordenados y retorna el primer itinerario ordenado o una lista vacía si no hay resultados.

Tabla comparativa

		Tiempo de ejecución									
No. Vuelos	**No. Aeropuertos**	**Itinerarios**	**Itinerarios Par**	**Itinerarios Tiempo**	**Itinerarios TiempoPar**	**Itinerarios Aire**	**Itinerarios AirePar**	**Itinerarios Escala**	**Itinerarios EscalaPar**	**ISalida**	**IsalidaPar**
99	**22**	**3.8 ms**	**16.8 ms**	**13.0 ms**	**18.4 ms**	**19.0 ms**	**18. 2 ms**	**2.2 ms**	**7.0 ms**	**2.7 ms**	**6.1**

```
Unable to create a system terminal
Tiempo promedio de ejecuci◆n de itinerarios: 3.8862199999999993 ms
Tiempo promedio de ejecuci◆n de itinerariosPar: 16.848 ms

Tiempo promedio de ejecuci◆n de itinerariosTiempo: 13.0131 ms
Tiempo promedio de ejecuci◆n de itinerariosTiempoPar: 18.4831 ms

Tiempo promedio de ejecuci◆n de itinerariosEscalas: 2.2743 ms
Tiempo promedio de ejecuci◆n de itinerariosEscalasPar: 7.0323 ms

Tiempo promedio de ejecuci◆n de itinerariosAire: 19.0129 ms
Tiempo promedio de ejecuci◆n de itinerariosAirePar: 18.2427 ms

Tiempo promedio de ejecuci◆n de itinerariosSalida: 2.7815 ms
Tiempo promedio de ejecuci◆n de itinerariosSalidaPar: 6.1057 ms
```

Conclusiones

En la tabla comparativa podemos apreciar que el índice de ganancia en el tiempo de ejecución de los itinerarios y los itinerariosPar son bastante disparejos. En el métodos de itinerarios e itinerariosPar no hay una ganancia, debido que la función usa la recursión de cola. Como consecuencia, tenemos que su recursividad depende de el anterior valor, por lo que, la forma secuencial se vuelve un poco más eficiente al momento de buscar la lista de vuelos.